

Laboratory work 4: **Operational Systems**

Elaborated:
st. gr. FAF-213

Afteni Maria,
Bajenov Sevastian,
Konjevic Alexandra

Verified:

Rastislav Călin

Chișinău – 2023

Topic: Boot Loader

Objectives: Create in assembly language an application that will act as a Boot Loader and will perform the following tasks:

1. It will display a greeting message that includes the author's name and will await keyboard input for the 'source' address on the floppy disk, from where the kernel (or other compiled code intended to be loaded and executed) will be read. The address should be entered in the SIDE, TRACK, SECTOR format and must strictly fall within the range reserved for the author student, as it was in Lab3.
2. It will wait for keyboard input for the 'destination' address in RAM where the data block read from the floppy disk will be loaded. The RAM address should be in the format XXXXh:XXXXh, identical to how it was in Lab3.
3. It will transfer the FLOPPY ==> RAM data and display the error code with which the operation was completed.
4. It will display a message to press a key to launch the kernel (or execute the desired code).
5. Upon completion of the kernel execution or executed code, it will display a message to press a key to repeatedly execute the Boot Loader.

Implementation:

To create a boot loader system capable of loading a kernel or other compiled code from the floppy disk into RAM, and then, executing it, there were created two bootloaders: ``bootloader.asm`` and ``mini_boot.asm``. The first part (``bootloader.asm``) contains the main functionality, while the second part (``mini_boot.asm``) is a small boot loader meant to load the main boot loader.

The code begins with the `org `7d00h`` directive, setting the origin point where the bootloader code will reside in memory.

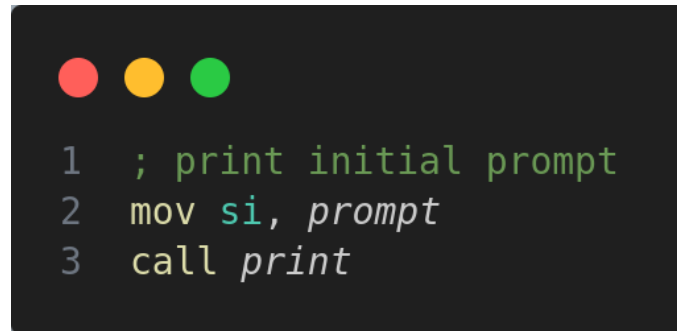
Various data sections (``section .data`` and ``section .bss``) are declared to hold prompts, user inputs, and intermediate variables:

```
1 section .data:
2     prompt db 'Welcome, Alexandra. Press any key to continue: $'
3     hts_prompt db "Enter CHTS script address$"
4     so_prompt db "Enter XXXX:YYYY RAM address$"
5     kernel_start db "Press any key to load the kernel: $"
6
7     sc db 0
8     h db 0
9     t db 0
10    s db 0
11
12    c db 0
13    result db 0
14
15    marker db 0
16
17 section .bss:
18     hex_result resb 2
19     add1 resb 2
20     add2 resb 2
21     buffer resb 2
22
```

The line that follows after this - ``dw 0AA55h`` - is marking the end of a boot sector, indicating

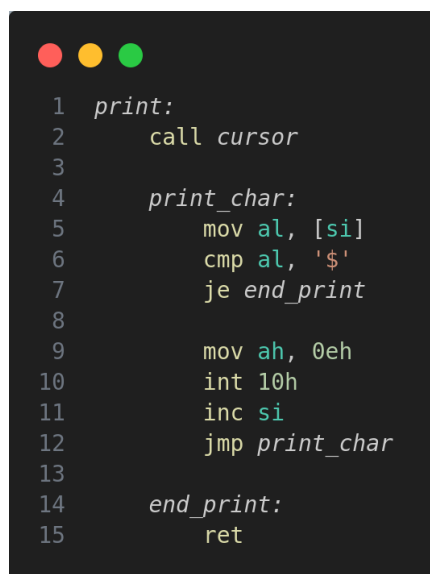
to the BIOS that the sector is bootable and the code contained within it should be executed during the boot process.

The next step is displaying the initial prompt(*`Welcome, Alexandra. Press any key to continue: \$`*):




```
1 ; print initial prompt
2 mov si, prompt
3 call print
```

The function *`print`* is being called:



```
1 print:
2     call cursor
3
4     print_char:
5         mov al, [si]
6         cmp al, '$'
7         je end_print
8
9         mov ah, 0eh
10        int 10h
11        inc si
12        jmp print_char
13
14    end_print:
15        ret
```

This function also calls another function: *`cursor`* which sets the position of the cursor using the function *`03h`* of the interruption *`10h`*:



```
1 cursor:
2     mov ah, 03h
3     mov bh, 0
4     int 10h
5
6     ret
7
```

Next, the printing function declares the *`print_char`* label, which loads the character from the memory address pointed to by *`si`* into the *`al`* register and compares the character loaded with the ASCII value of *`\$`*. This symbol *`\$`* marks the end of the prompt. If the character loaded is *`\$`*, it jumps to the *`end_print`* label, which effectively ends the printing process.

Printing characters occurs in the following mode: the function sets up the video display function

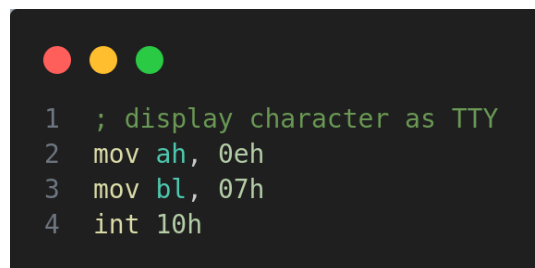
(`0eh`) of BIOS interrupt `10h` for displaying characters. After that, there is incremented the memory address in `si` to point to the next character in the string and the function jumps back to the `print_char` label to continue printing the next character.

After the prompts are displayed, the user input is read, using the function `00h` of the interruption `16h`:



```
1 ; read option
2 mov ah, 00h
3 int 16h
```

And this character is displayed:



```
1 ; display character as TTY
2 mov ah, 0eh
3 mov bl, 07h
4 int 10h
```

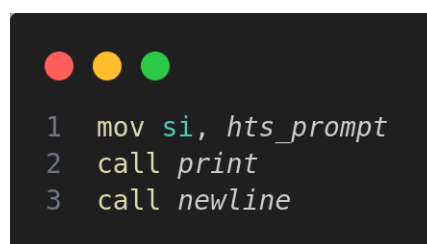
After that, there is printed a new line, by calling the function `newline`:



```
1 newline:
2     call cursor
3
4     mov ah, 02h
5     mov bh, 0
6     inc dh
7     mov dl, 0
8     int 10h
9
10    ret
```

This function sets the position of the cursor on a new line.

Next, the next prompt is displayed, to get the CHTS script address (`Enter CHTS script address$`):



```
1 mov si, hts_prompt
2 call print
3 call newline
```

The next block of code is responsible for displaying the `>` symbol in front of the user input:

```

1 ; print sector count prompt
2 mov ah, 0eh
3 mov al, '>'
4 mov bl, 07h
5 int 10h

```

Next, we have the instruction that moves the value 0 into the memory location pointed to by the label `[result]`. After that, the functions `clear` and `read_buffer` are called:

```

1 clear:
2     mov byte [c], 0
3     mov byte [si], 0
4     mov si, buffer
5
6     ret

```

The clear function sets the value 0 to the memory locations pointed by `c` and `si`, and moves the value of the `buffer` to the register `si`.

Next, we have the `read_buffer` function:

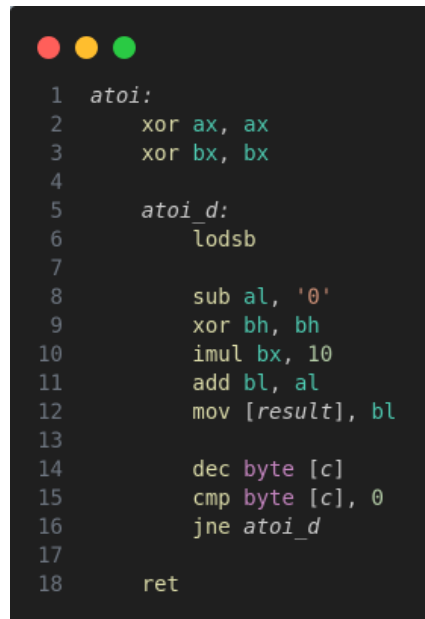
```

1 read_buffer:
2
3     read_char:
4         ; read character
5         mov ah, 00h
6         int 16h
7
8         ; check if the ENTER key was introduced
9         cmp al, 0dh
10        je hdl_enter
11
12        ; check if the BACKSPACE key was introduced
13        cmp al, 08h
14        je hdl_backspace
15
16        ; add character into the buffer and increment its pointer
17        mov [si], al
18        inc si
19        inc byte [c]
20
21        ; display character as TTY
22        mov ah, 0eh
23        mov bl, 07h
24        int 10h
25
26        jmp read_char
27
28    hdl_enter:
29        mov byte [si], 0
30        mov si, buffer
31
32        cmp byte [marker], 0
33        je atoi_jump
34        jmp atoh_jump
35
36    hdl_backspace:
37        call cursor
38
39        cmp byte [c], 0
40        je read_char
41
42        ; clear last buffer char
43        dec si
44        dec byte [c]
45
46        ; move cursor to the left
47        mov ah, 02h
48        mov bh, 0
49        dec dl
50        int 10h
51
52        ; print space instead of the cleared char
53        mov ah, 0ah
54        mov al, ' '
55        mov bh, 0
56        mov cx, 1
57        int 10h
58
59        jmp read_char
60
61    atoi_jump:
62        call atoi
63        jmp end_read_buffer
64
65    atoh_jump:
66        call atoh
67        jmp end_read_buffer
68
69    end_read_buffer:
70
71    ret

```

First of all, it reads a character from the keyboard (with function `00h` of the `16h` interruption), after that, handles the enter or backspace keys. If the keys are neither of them, then the introduced character is added to the buffer, and the buffer pointer is incremented. The character read from the keyboard is also displayed on the screen.

The handling of the enter key uses also additional helper functions:

A screenshot of a code editor showing assembly code for the `atoi` function. The code is written in a dark-themed editor with syntax highlighting. It starts with a label `atoi:` at line 1, followed by instructions to zero out the `ax` and `bx` registers. A loop label `atoi_d:` is at line 5. Inside the loop, a byte is loaded from memory into `al`, zeroed out, multiplied by 10 in `bx`, and added to `bx`. The result is stored in `result`. The memory pointer `c` is decremented and compared to zero. If not zero, the loop continues. The function ends with `ret` at line 18.

```
1  atoi:
2      xor ax, ax
3      xor bx, bx
4
5      atoi_d:
6          lodsb
7
8          sub al, '0'
9          xor bh, bh
10         imul bx, 10
11         add bl, al
12         mov [result], bl
13
14         dec byte [c]
15         cmp byte [c], 0
16         jne atoi_d
17
18     ret
```

This function is responsible for converting a sequence of ASCII characters representing a decimal number into its corresponding numerical value. The instructions `xor ax, ax` and `xor ab, ab` set the values of this registers to zero. `atoi_d` label marks the beginning of the loop where the actual conversion takes place. `lods b`: This instruction loads the byte addressed by the `si` register into the `al` register and increments `si` automatically, this means it loads the next character from memory into `al` on each iteration.

Conversion steps: `sub al, '0'` subtracts the ASCII value of character '0' from the ASCII value of the current character in `al`. ASCII values are numeric representations of characters, and subtracting the ASCII value of '0' effectively converts the character to its numerical value. Next, the `bh` register is cleared. `imul bx, 10` multiplies the value in `bx` (which initially holds the previously processed part of the number) by 10. This prepares for the next digit to be added. `imul` stands for "integer multiply". It effectively multiplies BX by 10 and stores the result back in `bx`. `add bl, al` adds the converted numerical value in `al` to the `bl`. It accumulates the digits to form the final integer value. Next, this value is stored into the memory location labeled as `result`. This is the accumulating result of the conversion. `dec byte [c]` decrements the byte in memory pointed to by `c`. `cmp byte [c], 0`: Compares the value pointed to by `c` with zero to check if the conversion is complete. If there are more characters to convert (`[c]` is not zero), the loop continues (`jne atoi_d`) to process the next character. If all characters have been processed, the subroutine proceeds to `ret`, indicating the end of the conversion.

Also, the handling of enter uses the function:

```

1  atoh:
2      xor bx, bx
3      mov di, hex_result
4
5      atoh_s:
6          xor ax, ax
7          mov al, [si]
8
9          cmp al, 65
10         jg atoh_l
11         sub al, 48
12         jmp continue
13
14         atoh_l:
15             sub al, 55
16             jmp continue
17
18         continue:
19             mov bx, [di]
20             imul bx, 16
21             add bx, ax
22             mov [di], bx
23
24             inc si
25
26             dec byte [c]
27             jnz atoh_s
28
29         ret

```

The `atoh` (ASCII to Hexadecimal) subroutine is responsible for converting a sequence of ASCII characters representing a hexadecimal number into its corresponding numerical value. First of all, it clears the value of register `bx` and sets the value of register `di` to the value of `hex_result`. `atoh_s` label marks the beginning of the loop where the actual conversion takes place.

`mov al, [si]` loads the byte addressed by the `si` register into the `al` register. This retrieves the next character from memory for processing. After that, if the value from `al` is greater than 65, the code jumps to `atoh_l`. If the character is less than or equal to '9', it proceeds directly to the next step. `jg atoh_l` (if the character is greater than 'A', it means the character is in the range 'A' to 'F').

The function subtracts 55 from the ASCII value (`al`), converting characters 'A' to 'F' into their respective hexadecimal values ('A' = 10, 'B' = 11, ..., 'F' = 15). `sub al, 48`: If the character is '0' to '9', subtracts 48 from the ASCII value (`al`) - this converts ASCII characters '0' to '9' into their respective numerical values (0 to 9).

`continue` label moves the current value in `bx` into memory location `[di]` (`hex_result`); multiplies the value in `bx` by 16 to shift it left by one hexadecimal digit. After that, it adds the converted value in `ax` to `bx` and stores the updated value in `[di]`. It then increments the `si` register to move to the next character and decrements the byte in memory pointed to by `c` to track the length of the input string. If there are more characters to convert (`[ci]` is not zero), the loop continues (`jnz atoh_s`).

The handling of the backspace is done this way:

```

1  hdl_backspace:
2      call cursor
3
4      cmp byte [c], 0
5      je read_char
6
7      ; clear last buffer char
8      dec si
9      dec byte [c]
10
11     ; move cursor to the left
12     mov ah, 02h
13     mov bh, 0
14     dec dl
15     int 10h
16
17     ; print space instead of the cleared char
18     mov ah, 0ah
19     mov al, ' '
20     mov bh, 0
21     mov cx, 1
22     int 10h
23
24     jmp read_char

```

The `hdl_backspace` segment checks if there's a character in the buffer to be deleted. If there's a character present, it erases the last character in the buffer, moves the cursor back one position on the screen, prints a space character to visually erase the character on the display, and then resumes waiting for new input by jumping back to the read character section. If there are no characters in the buffer, it continues to wait for new input without any erasing action.

After the value of sector count is read, and the functions `clear` and `read_buffer` are called, the same steps are used to read the value of head, track, sector, ram address, segment and offset.

After the buffer is read, the register `ax` is set to the value of `hex_result`, and the value from `ax` is moved to the `[add2]`.

```

1  mov ax, [hex_result]
2  mov [add2], ax

```

After that, the function `load_kernel` is called:

```

1  load_kernel:
2      mov ah, 0h
3      int 13h
4
5      mov ax, [add2]
6      mov es, ax
7      mov bx, [add1]
8
9      ; load the NASM script into memory
10     mov ah, 02h
11     mov al, [sc]
12     mov ch, [t]
13     mov cl, [s]
14     mov dh, [h]
15     mov dl, 0
16
17     int 13h
18
19     ; print error code
20     mov al, '0'
21     add al, ah
22     mov ah, 0eh
23     int 10h
24
25     call newline
26
27     ret

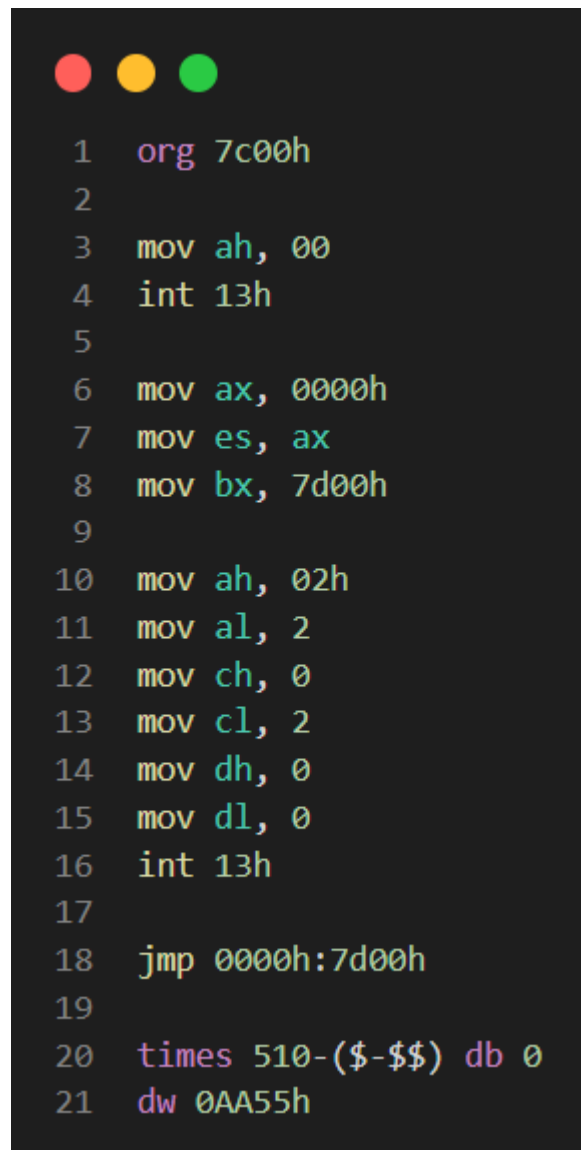
```

`mov ah, 0h` sets `ah` register to 0, indicating the "reset disk system" function for the BIOS interrupt. `mov ax, [add2]` moves the value stored in memory at location `add2` into the `ax`

register. The function ``02h`` of the interruption ``13h`` is executed, and the values of the ``sector_count``, ``track``, ``sector``, ``head`` are read into the memory at ``ES:BX``. After that, the error code is displayed on the screen.

After the ``load_kernel`` function is executed, the next sequence of code displays a prompt asking the user to load a kernel, waits for the user to input a key, displays the input as a character on the screen, and then combines and jumps to the memory address formed by the contents of ``add1`` and ``add2``.

``mini_boot.asm``:



```
1  org 7c00h
2
3  mov ah, 00
4  int 13h
5
6  mov ax, 0000h
7  mov es, ax
8  mov bx, 7d00h
9
10 mov ah, 02h
11 mov al, 2
12 mov ch, 0
13 mov cl, 2
14 mov dh, 0
15 mov dl, 0
16 int 13h
17
18 jmp 0000h:7d00h
19
20 times 510-($-$$) db 0
21 dw 0AA55h
```

``mini_boot.asm`` is a smaller boot loader that loads the main boot loader (``bootloader.asm``) and jumps to its entry point. The code follows a sequence of steps:

- Display prompts to the user for input (source address, destination address in RAM).
- Read input and handle user interactions (keypresses, backspaces).
- Convert ASCII input to usable numeric values (e.g., converting ASCII input for address to its actual hex value).

- Load the kernel from the floppy disk to the specified RAM address.
- Display messages and prompts at various stages.
- Use BIOS interrupt calls (int 10h, int 13h, etc.) to handle input/output and disk operations.

Running the code

In order to create a bootable image for the virtual machine, first of all we need to assemble the program from the file with the individual task, using the command `nasm task.asm`. This command creates a binary file. After that, we need to execute the script `boot_floppy_script.sh`, passing it three parameters – the binary file, the bootloader (`bootloader.asm`) and the second bootloader (`mini_boot.asm`): `../boot_floppy_script.sh rainbow bootloader.asm mini_boot.asm`. The script is the following:

```

1  #!/bin/bash
2
3  binary_file="$1"
4  bootloader="$2"
5  mini_bootloader="$3"
6
7  # Check if the binary file exists
8  if [ ! -f "$binary_file" ]; then
9      echo "Error: Binary file '$binary_file' does not exist."
10     exit 1
11 fi
12
13 nasm -f bin $mini_bootloader -o mini_bootloader.bin
14 nasm -f bin $bootloader -o bootloader.bin
15
16 # Create an empty floppy disk image (1.44MB size)
17 floppy_image="floppy.img"
18 truncate -s 1474560 mini_bootloader.bin
19 mv mini_bootloader.bin $floppy_image
20
21 dd if="bootloader.bin" of="$floppy_image" bs=512 seek=1 conv=notrunc
22 dd if="$binary_file" of="$floppy_image" bs=512 seek=3 conv=notrunc
23
24 echo "Binary file '$binary_file' successfully added to floppy image '$floppy_image'."
25

```

The script takes three arguments from the command line: \$1 (binary_file), \$2 (bootloader), and \$3 (mini_bootloader). It checks if the binary file provided as \$1 exists. If it doesn't exist, it displays an error message and exits the script with a non-zero status. It uses NASM with the `-f bin` option to assemble the `mini_bootloader` file into `mini_bootloader.bin`. Similarly, it assembles the `bootloader` file into `bootloader.bin`.

After that, the script creates an empty floppy disk image named `floppy.img` with a size of 1.44MB using the `truncate` command. It then is essentially moving the `mini_bootloader.bin` file and renaming it to the name stored in the variable `$floppy_image` (which is `floppy.img` as per the script).

Using `dd` (data duplicator), it writes the `bootloader.bin` content to the created floppy image starting at the second block (512 bytes per block) using `seek=1`. This places the bootloader in the boot sector. Similarly, it writes the content of the binary file (`$binary_file`) to the floppy image starting at the fourth block (`seek=3`). This presumably adds the program or kernel to the floppy image.

Conclusion:

In conclusion, the laboratory work involving the implementation of the provided code has provided a valuable learning experience in low-level programming,

bootloader development, and BIOS interrupt handling. It underscores the importance of meticulousness, thorough testing, and a deep understanding of hardware interactions in the realm of systems programming.